CS477 - Formal Software Development Methods

Formal Definition of a Simple Programming Language

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Formal Operational Semantics

By a *formal operational semantics* of a programming language, one typically understands a collection of axioms or rules, typically in some rigorous logical formalism, either specifying how its expressions, statements, programs, etc., are evaluated/executed, or specifying sufficient constraints over their evaluation/execution to be able to unambiguously mechanically infer to what values they evaluate to (in case their evaluation terminates).

These collections of axioms or rules are called an "operational semantics" because they say how a possible implementation of a programming language should "operate". In general, it is not difficult in practice to give an implementation of (an interpreter of) a language in any programming languages by just following and translating its operational semantics into the target implementation language.

There is no definite agreement on how an operational semantics of a language should be given, because any description of a programming language which is rigorous enough to quickly lead to a correct implementation of the language can be considered to be a valid operational semantics. There are scientists who consider that even an adhoc implementation of a language interpreter is an operational semantics. The advantage of giving a language a formal operational semantics in a logical formalism as opposed to just implementing an adhoc interpreter is that one can also *formally reason about programs*, not only to execute them.

In this lecture we show how one can define a simple programming language as an equational theory whose equations can be also regarded as rewrite rules and thus obtain a formal operational semantics of the programming language. We will define several other languages in this class following a similar style.

Syntax of a Simple Language

We will exemplify our equational language definitional style by means of a very simple non-procedural imperative language which has arithmetic and boolean expressions, conditionals and while loops.

A program is a sequence of statements followed by an expression. The expression is evaluated in the state obtained after evaluating all the statements and its result is returned as the result of the evaluation of the entire program. Arithmetic and boolean expressions do not have side effects; only statements can change the state.

Formally, the syntax of this simple language can be given as a context-free grammar (CFG) as follows:

$$
\begin{array}{rcl}
Var & ::= & \text{standard identifiers} \\
AExp & ::= & Var \mid 1 \mid 2 \mid 3 \mid ... \mid \\
 & & AExp + AExp \mid AExp - AExp \mid AExp * AExp \mid AExp / AExp \\
BExp & ::= & \mathsf{true} \mid \mathsf{false} \mid AExp \leq AExp \mid AExp \geq AExp \mid AExp == AExp \\
 & & BExp \ \mathsf{and} \ BExp \mid BExp \ \mathsf{or} \ BExp \mid \mathsf{not} \ BExp \\
Stmt & ::= & \mathsf{skip} \mid Var := AExp \mid Stmt \ ; \ Stmt \mid \{Stmt\} \\
 & & \mathsf{if} \ BExp \ \mathsf{then} \ Stmt \ \mathsf{else} \ Stmt \mid \mathsf{while} \ BExp \ Stmt \\
Pgm & ::= & Stmt \ ; \ AExp
\end{array}
$$

Defining Syntax as an Algebraic Signature

As mentioned in previous lectures, the mix-fix notation for algebraic signatures is equivalent in expressivity to CFGs. Let us next see how we can formalize the syntax of our simple programming language as a signature in Maude. We next define a module without any equations or rules, its only role being to define the syntax of our language. Please compare the signature below to the CFG above.

We want our language to use Maude's *builtin libraries* of integers (as special arithmetic expressions) and identifiers (as variables), to avoid redefining them and thus focus on the more interesting aspects of our language definitions (nevertheless, one is free to define everything from scratch, including ones own version of integers and identifiers).

Therefore, we start by including the builtin modules `INT` and `QID`:

```
fmod SYNTAX is
  including INT .
  including QID .
```

The identifiers in `QID` are quoted: `'a`, `'abc`, `'a1`, `'x17`, etc. We next define the syntax of our variables: any quoted identifier is allowed to be a variable in our language, plus all the one letter "unquoted" identifiers; to achieve the latter, we define them as constants of sort `Var`:

```
--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var .
```

We next define the syntax of arithmetic expressions. Recall that both variables and integers are arithmetic expressions. To avoid syntactic clashes with operations with the same name (addition, multiplication, etc.) already defined on integers, we prefer to "wrap" integers with an operator "`#`":

```
--- AExp
  sort AExp .
  subsort Var < AExp .
  op #_ : Int -> AExp [prec 30] .
  ops _+_ _-_ : AExp AExp -> AExp [prec 33 gather (E e)] .
  ops _*_ _/_ : AExp AExp -> AExp [prec 31 gather (E e)] .
```

Note that addition and subtraction have the same precedence, and also multiplication and division; all these operators are parsed as left associative, because of the attribute "`gather (E e)`".

Syntax for boolean expressions:

```
--- BExp
  sort BExp .
  ops true false : -> BExp .
  ops _<=_ _>=_ _==_ : AExp AExp -> BExp [prec 37] .
  op _and_ : BExp BExp -> BExp [prec 55] .
  op _or_ : BExp BExp -> BExp [prec 57] .
  op not_ : BExp -> BExp [prec 53] .
```

Syntax for statements:

```
--- Stmt
  sort Stmt .
  op skip : -> Stmt .
  op _:=_ : Var AExp -> Stmt [prec 40] .
  op _;_ : Stmt Stmt -> Stmt [gather (e E) prec 70] .
  op {_} : Stmt -> Stmt .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 60] .
  op while__ : BExp Stmt -> Stmt [prec 60] .
```

Syntax for programs:

```
--- Pgm
  sort Pgm .
  op _;_ : Stmt AExp -> Pgm [prec 80] .
endfm
```

We are done with defining the syntax of our simple language. It is good methodology to keep the definition of the syntax of a language separate from the definition of its semantics. Let us then save the syntactic definition above into a separate file, say `syntax.maude`.

(All the Maude definitions/files discussed in class are provided as well on the website).

Parsing Programs

Now that the syntax is finished, we can write and parse programs. One may want to set the "print with parentheses" flag on, to better see the abstract syntax trees (AST); if one to see the ASTs in the more usual prefix form, then one can also set the mix-fix flag off:

```
set print with parentheses on .
parse x .

parse 'x + # 1 .

parse x - # 10 .

parse 'x := # 10 .

parse 'x := # 10 + 'x .
```

```
parse x + # 2 := # 10 .

parse x := # 1 ; y := x .

parse skip ; # 3 + y .

parse x := # 1 ; y := x ; y .

parse x := # 1 ; y + # 1 := x ; y .

parse
      x := # 1 ;
      y := (# 1 + x) * # 2 ;
      z := x * # 2 + x * y + y * # 2 ;
      x + y + z
 .
```

The following program calculates x^y:

```
parse
      x := # 17 ;
      y := # 100 ;
      p := # 1 ;
      i := y ;
      while not i == # 0 {
        p := p * x ;
        i := i - # 1
      } ;
      p
 .
```

The following program calculates the n^{th} Fibonacci's number:

```
parse
      x := # 0 ;
      y := # 1 ;
      n := # 1000 ;
      i := # 0 ;
      while not i == n {
        y := y + x ;
        x := y - x ;
        i := i + # 1
      } ;
      y
 .
```

The following program tests Collatz' conjecture for a given n:

```
parse
      n := # 178378342647823759743985734809582309829798347583490
      c := # 0 ;
      while not n == # 1 {
        c := c + # 1 ;
        if n == # 2 * (n / # 2)
        then n := n / # 2
        else n := # 3 * n + # 1
      } ;
      c
  .
```

We can now replace the keyword "`parse`" by "`rewrite`" and place all these programs in a file, say `programs-rewrite.maude`, that we will use later to test our subsequent semantic definition.

State

For this simple imperative language, a *state* is a map from variables to integer numbers $\texttt{Var} \to \texttt{Int}$. We let σ, σ', etc., denote states. If σ is a state and x a variable, then we let $\sigma[x]$ or $\sigma(x)$ denote the integer value to which σ maps x. Moreover, if x is a variable and i an integer, then we let $\sigma[x \leftarrow i]$ denote the function $\texttt{Var} \to \texttt{Int}$ defined as follows:

$$\sigma[x \leftarrow i](y) = \begin{cases} \sigma(y) & \text{if } x \neq y, \\ i & \text{if } x = y. \end{cases}$$

We also let $\emptyset$ denote the initial state. For simplicity, we here assume that all variables are initialized with 0 at the beginning of the computation. In other words, we consider that $\emptyset[x]$ is 0. Another possibility would be to consider states as *partial* functions and thus let $\emptyset[x]$ undefined, but this would create more cases to analyze in the subsequent definition.

Defining the State in Maude

As said above, for our simple language a state is nothing but a mapping of variables into values. This can be defined many different and equivalent ways, for example as a set of pairs "variable = integer".

We first define the constructors of such mappings:

```
fmod STATE is
  including SYNTAX .
  sort State .
  op _=_ : Var Int -> State .
  op __ : State State -> State [assoc comm id: empty ] .
  op empty : -> State .
```

Next we define the two operations on states, namely variable lookup and update:

```
  var S : State . var V : Var . var I I' : Int .

  --- retrieves a value from the state
  op _[_] : State Var -> Int .
  eq (S V = I)[V] = I .
  eq S[V] = 0 [owise] . --- default value 0

  --- updates a variable in the state
  op _[_<-_] : State Var Int -> State .
  eq (S V = I)[V <- I'] = S V = I' .
  eq S[V <- I] = S V = I [owise] .
endfm
```

Semantics

The next semantic approach to define a programming language is the most straightforward rewrite-based, or equational, style to define a programming language. Assuming one's familiarity with term rewriting and/or equational reasoning, then one would most likely define a (simple) programming language following this style.

The idea is to define a rewrite-based "implementation" of a function

$$\textit{(Syntactic Category)} \times \textit{State} \rightarrow \textit{Result}$$

that takes a term in any syntactic category, such as an arithmetic or a boolean expression or a statement or a program in our case, and a state, and returns the result obtained after evaluating the given term in the given state.

This would also be the style followed by one who wants to implement rapidly an interpreter for a (simple) language.

Configuration

One can think of an operational semantics as one that defines a transition relation on *configurations*. In general, a configuration is a tuple containing a term over the syntax of the language and corresponding needed semantic infrastructure, such as a state, various control stacks, etc.; however, in our simple language definition we only need configurations consisting of pairs of a term and a state.

In our simple language definition, we only need simple *configurations*. We enclose the various components forming a configuration with angle brackets. For example, $\langle a, \sigma \rangle$ is a configuration containing an arithmetic expression a and a state σ, and $\langle b, \sigma \rangle$ is a configuration containing a boolean expression b and a state σ.

Configuration of different types need not necessarily have the same number of components. For example, since all programs evaluate in the initial state, there is no need to mention a state next to a program in a configuration; in this case, a configuration is simply a one element tuple, $\langle p \rangle$, where p is a program.

Defining Configurations in Maude

Thanks to the mix-fix parsing capabilities of Maude, we can use the angled notation for configurations constructors:

```
fmod CONFIGURATION is
  including STATE .
  op <_,_> : AExp State -> Int .
  op <_,_> : BExp State -> Bool .
  op <_,_> : Stmt State -> State .
  op <_> : Pgm -> Int .
endfm
```

A term `< E,S >` is thought of as a "function call" that evaluates `E` in state `S`.

Semantics of Arithmetic Expressions

Having this functional approach in mind, we can now trivially define the evaluation of arithmetic expressions as follows:

```
fmod AEXP-RULES is
  including CONFIGURATION .
  var X : Var .  var S : State .
  var I : Int .  var E1 E2 : AExp .

  eq < X,S > = S[X] .
  eq < # I,S > = I .
  eq < E1 + E2,S > = < E1,S > + < E2,S > .
  eq < E1 - E2,S > = < E1,S > - < E2,S > .
  eq < E1 * E2,S > = < E1,S > * < E2,S > .
  eq < E1 / E2,S > = < E1,S > quo < E2,S > .
endfm
```

Semantics of Boolean Expressions

Next we define the evaluation of boolean expressions:

```
fmod BEXP-RULES is
  including CONFIGURATION .
  var S : State . var A1 A2 : AExp . var B B1 B2 : BExp .
  eq < true,S > = true .
  eq < false,S > = false .
  eq < A1 <= A2,S > = < A1,S > <= < A2,S > .
  eq < A1 >= A2,S > = < A1,S > >= < A2,S > .
  eq < A1 == A2,S > = < A1,S > == < A2,S > .
  eq < B1 and B2,S > = < B1,S > and < B2,S > .
  eq < B1 or B2,S > = < B1,S > or < B2,S > .
  eq < not B,S > = not < B,S > .
endfm
```

Semantics of Statements

The evaluation of statements follows a similar approach:

```
fmod STMT-RULES is
  including CONFIGURATION .
  var S : State .  var X : Var .  var A : AExp .
  var St St1 St2 : Stmt .  var B : BExp .
  eq < skip,S > = S .
  eq < X := A,S > = S[X <- < A,S >] .
  eq < St1 ; St2,S > = < St2, < St1,S > > .
  eq < { St },S > = < St,S > .
  eq < if B then St1 else St2,S >
   = if < B,S > then < St1,S > else < St2,S > fi .
  eq < while B St,S >
   = if < B,S > then < St ; while B St,S > else S fi .
endfm
```

Semantics of Programs

Program evaluation is also straightforward:

```
fmod PGM-RULES is
  including CONFIGURATION .
  var St : Stmt . var A : AExp .
  eq < St ; A > = < A,< St,empty > > .
endfm

fmod FUNCTIONAL-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including STMT-RULES .
  including PGM-RULES .
endfm
```

One can now place all the semantic definition modules above into one Maude module, say `semantics.maude`.

Exercise 1 *Execute the provided Maude definition of the simple language discussed above. Also execute some other programs of your choice. Also, trace the execution of some programs (using first the Maude command "*`set trace on .`*" and then reducing the program) to see how each execution step corresponds to an equational deductive step.*

Homework Exercise

There is only one HW exercise in this lecture, which is described below. This is the most difficult HW exercise so far, so start early!

Homework Exercise 1 *You are required to add two language constructs to our simple language:*

- `inc : Var -> AExp`, *which evaluates to the value of the variable and then increments the value of that variable; this is like ++ in Java or C++; and*
- `halt : AExp -> Stmt`.

You are supposed to modify the provided Maude code. Hint: you will have to modify the provided semantics globally; your new configurations will need to evaluate to pairs containing a value and a state (because `inc` *has side effects); then the state will need to be propagated properly; for example, the equation for + will be*

something like:

```
eq < E1 + E2,S >
 = <
     val(< E1,S >) + val(< E2,state(< E1,S >) >),
     state(< E2,state(< E1,S >) >)
   > .
```

In the above, `val` *and* `state` *are projection operators that you will also need to define.*